# Formal Language Recognition with the Java Type Checker

**Yossi Gil[1] and Tomer Levy[2]**

1 **Department of Computer Science, The Technion—Israel Institute of Technology, Haifa, Israel.**
2 **Department of Computer Science, The Technion—Israel Institute of Technology, Haifa, Israel.**

*"Java generics are 100% pure syntactic sugar, and do not support meta-programming"*[1]

## Abstract

This paper is a theoretical study of a practical problem: the automatic generation of Java Fluent APIs from their specification. We explain why the problem's core lies with the expressive power of Java generics. Our main result is that automatic generation is possible whenever the specification is an instance of the set of deterministic context-free languages, a set which contains most "practical" languages. Other contributions include a collection of techniques and idioms of the limited meta-programming possible with Java generics, and an empirical measurement demonstrating that the runtime of the "javac" compiler of Java may be exponential in the program's length, even for programs composed of a handful of lines and which do not rely on overly complex use of generics.

**1998 ACM Subject Classification** D.3.2: Java, D.3.4 Processors: Parsing, D.3.2 Language classifications: Nonprocedural languages, Specialized application languages, F.4.2 Grammars and Other Rewriting Systems: Classes defined by grammars or automata, Classes defined by resource-bounded automata, F.1.1 Models of Computation: Automata

**Keywords and phrases** Parser Generators, Generic Programming, Fluent API

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2016.10

## 1 Introduction

Ever after their inception[2] *fluent APIs* increasingly gain popularity [20, 28, 31] and research interest [16,29]. In many ways, fluent APIs are a kind of *internal **D**omain **S**pecific **L**anguage*: They make it possible to enrich a host programming language without changing it. Advantages are many: base language tools (compiler, debugger, IDE, etc.) remain applicable, programmers are saved the trouble of learning a new syntax, etc. However, these advantages come at the cost of expressive power; in the words of Fowler: "*Internal DSLs are limited by the syntax and structure of your base language.*"[3]. Indeed, in languages such as C++ [37], fluent APIs often make extensive use of operator overloading (examine, e.g., Ara-Rat [23]), but this capability is not available in Java [4].

[1] Found on stackoverflow: `http://programmers.stackexchange.com/questions/95777/generic-programming-how-often-is-it-used-in-industry`
[2] `http://martinfowler.com/bliki/FluentInterface.html`
[3] M. Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?*, 2005 `http://www.martinfowler.com/articles/languageWorkbench.html#InternalDsl`

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 10; pp. 10:1–10:27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
from("direct:a").choice()
    .when(header("foo").isEqualTo("bar"))
      .to("direct:b")
    .when(header("foo").isEqualTo("cheese"
     ))
      .to("direct:c")
    .otherwise()
      .to("direct:d");
```

**(a)** Apache Camel

```
create
  .select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, count
     ())
  .from(AUTHOR)
  .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTHOR_ID))
  .where(BOOK.LANGUAGE.eq("DE"))
  .and(BOOK.PUBLISHED.gt(date("2008-01-01")))
  .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
  .having(count().gt(5))
  .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
  .limit(2)
  .offset(1);
```

**(b)** jOOQ

**Figure 1** Two examples of Java fluent API.

Despite this limitation, fluent APIs in Java can be rich and expressive, as demonstrated in Figure 1 showing use cases of the DSL of Apache Camel [27] (open-source integration framework), and that of jOOQ[4], a framework for writing SQL in Java, much like Linq [33].

Other examples of fluent APIs in Java are abundant: jMock [20], Hamcrest[5], EasyMock[6], jOOR[7], jRTF[8] and many more.

## 1.1 A Type Perspective on Fluent APIs

Figure 1(B) suggests that jOOQ imitates SQL, but, is it possible at all to produce a fluent API for the entire SQL language, or XPath, HTML, regular expressions, BNFs, EBNFs, etc.? Of course, with no operator overloading it is impossible to fully emulate tokens; method names though make a good substitute for tokens, as done in "`.when(header(foo).isEqualTo("bar")).`" (Figure 1). The questions that motivate this research are:

- Given a specification of a DSL, determine whether there exists a fluent API that can be made for this specification?
- In the cases that such fluent API is possible, can it be produced automatically?
- Is it feasible to produce a *compiler-compiler* such as Bison [17] to convert a language specification into a fluent API?

Inspired by the theory of formal languages and automata, this study explores what can be done with fluent APIs in Java.

Consider some fluent API (or DSL) specification, permitting only certain call chains and disallowing all others. Now, think of the formal language that defines the set of these permissible chains. We prove that there is always a Java type definition that *realizes* this fluent definition, provided that this language is *deterministic context-free*, where

- In saying that a type definition *realizes* a specification of fluent API, we mean that call chains that conform with the API definition compile correctly, and, conversely, call chains that are forbidden by the API definition do not type-check, resulting in an appropriate compiler error.

[4] http://www.jooq.org
[5] http://hamcrest.org/JavaHamcrest/
[6] http://easymock.org/
[7] https://github.com/jOOQ/jOOR
[8] https://github.com/ullenboom/jrtf

- Roughly speaking, deterministic context-free languages are those context-free languages that can be recognized by an LR parser[9] [2].
  An important property of this family is that none of its members is ambiguous. Also, it is generally believed that most practical programming languages are deterministic context-free.

A problem related to that of recognizing a formal language, is that of parsing, i.e., creating, for input which is within the language, a parse tree according to the language's grammar. In the domain of fluent APIs, the distinction between recognition and parsing is in fact the distinction between compile time and runtime. Before a program is run, the compiler checks whether the fluent API call is legal, and code completion tools will only suggest legal extensions of a current call chain.

In contrast, a parse tree can only be created at runtime. Some fluent API definitions create the parse-tree iteratively, where each method invocations in the call chain adding more components to this tree. However, it is always possible to generate this tree in "batch" mode: This is done by maintaining a *fluent-call-list* which starts empty and grows at runtime by having each method invoked add to it a record storing the method's name and values of its parameters. The list is completed at the end of the fluent-call-list, at which point it is fed to an appropriate parser that converts it into a parse tree (or even an AST).

## 1.2 Contribution

The answers we provide for the three questions above are:

1. If the DSL specification is that of a deterministic context-free language, then a fluent API exists for the language, but we do not know whether such a fluent API exists for more general languages.
   Recall that there are universal cubic time parsing algorithms [13,18,40] which can parse (and recognize) any context-free language. What we do not know is whether algorithms of this sort can be encoded within the framework of the Java type system.
2. There exists an algorithm to generate a fluent API that realizes any deterministic context-free languages. Moreover, this fluent API can create at runtime, a parse tree for the given language. This parse tree can then be supplied as input to the library that implements the language's semantics.
3. Unfortunately, a general purpose compiler-compiler is not yet feasible with the current algorithm.
   - One difficulty is usual in the fields of formal languages: The algorithm is complicated and relies on modules implementing complicated theoretical results, which, to the best of our knowledge, have never been implemented.
   - Another difficulty is that a certain design decision in the implementation of the standard `javac` compiler is likely to make it choke on the Java code generated by the algorithm.

Other concrete contributions made by this work include

- the understanding that the definition of fluent APIs is analogous to the definition of a formal language.
- a lower bound (deterministic pushdown automata) on the theoretical "computational complexity" of the Java type system.

[9] The "L" means reading the input left to right; the "R" stands for rightmost derivation

- an algorithm for producing a fluent API for deterministic context-free languages.
- a collection of generic programming techniques, developed towards this algorithm.
- a demonstration that the runtime of Oracle's `javac` compiler may be exponential in the program size.

## 1.3 Related Work

It has long been known that C++ templates are Turing complete in the following precise sense:

▶ Proposition 1. For every Turing machine, $m$, there exists a C++ program, $C_m$ such that compilation of $C_m$ terminates if and only if Turing-machine $m$ halts. Furthermore, program $C_m$ can be effectively generated from $m$ [25].

Intuitively, this is due to the fact that templates in C++ feature both recursive invocation and conditionals (in the form of "*template specialization*").

In the same fashion, it should be mundane to make the judgment that Java's generics are not Turing-complete since they offer no conditionals. Still, even though there are time complexity results regarding type systems in functional languages, we failed to find similar claims for Java.

Specialization, conditionals, `typedef`s and other features of C++ templates, gave rise to many advancements in template/generic/generative programming in the language [5, 7, 15, 34], including e.g., applications in numeric libraries [38, 39], symbolic derivation [22] and a full blown template library [1].

Garcia et al. [21] compared the expressive power of generics in half a dozen major programming languages. In several ways, the Java approach [11] did not rank as well as others.

Not surprisingly, work on meta-programming using Java generics, research concentrating on other means for enriching the language, most importantly annotations [36].

The work on SugarJ [19] is only one of many other attempts to achieve the embedded DSL effect of fluent APIs by language extensions.

Suggestions for semi-automatic generation can be found in the work of Bodden [10] and on numerous locations in the web. None of these materialized into an algorithm or analysis of complexity. However, there is a software artifact (fluflu[10]) that automatically generates a fluent API that obeys the transitions of a given finite automaton.

**Outline.** Section 2 is a brief reminder of method chaining, and fluent APIs, accompanied a discussion of how this work is related to type states. It is followed by a similar reminder of context-free languages, pushdown automata, and such in Section 3. Based on the vocabulary established this far, the main result is stated in Section 4.

Towards the proof in Section 7, Section 5 shows idioms and techniques for encoding computation with the Java type-checker. Section 6 makes use of these for encoding "jump-stack", a non-trivial data-structure, which is used, with suitable modifications, in the proof.

In Section 9, we discuss the challenges in translating the proof into a compiler-compiler for fluent APIs. In particular, this section demonstrates our claim (that may be surprising to some) that the standard Java compiler may spend an exponential time on compiling rather simple programs. Section 10 concludes with directions for further research.

[10] `https://github.com/verhas/fluflu`

```
String time(int hours, int minutes, int seconds)
      {
  final StringBuilder sb = new StringBuilder();
  sb.append(hours);
  sb.append(':');
  sb.append(minutes);
  sb.append(':');
  sb.append(seconds);
  return sb.toString();
}
```

**(a)** before

```
String time(int hours, int minutes, int seconds)
      {
    return new StringBuilder()
      .append(hours).append(':')
      .append(minutes).append(':')
      .append(seconds)
      .toString();
}
```

**(b)** after

**Figure 2** Recurring invocations of the pattern "invoke function on the same receiver", before, and after method chaining.

## 2 Method Chaining, Fluent APIs, and, Type States

The pattern "invoke function on variable **sb**", specifically with a function named **append**, occurs six times in the code in Figure 2(a), designed to format a clock reading, given as integers hours, minutes and seconds.

Some languages, e.g., SMALLTALK [24] offer syntactic sugar, called *cascading*, for abbreviating this pattern. *Method chaining* is a "programmer made" syntactic sugar serving the same purpose: If a method $f$ returns its receiver, i.e., **this**, then, instead of the series of two commands: `o.f(); o.g();`, clients can write only one: `o.f().g();`. Figure 2(b) is the method chaining (also, shorter and arguably clearer) version of Figure 2(a). It is made possible thanks to the designer of class **StringBuilder** ensuring that all overloaded variants of **append** return their receiver.

The distinction between *fluent API* and method chaining is the identity of the receiver: In method chaining, all methods are invoked on the same object, whereas in fluent API the receiver of each method in the chain may be arbitrary. Fluent APIs are more interesting for this reason. Consider, e.g., the following JAVA code fragment (drawn from JMock [20])

```
allowing(any(Object.class)).method("get.*").withNoArguments();
```

Let the return type of function **allowing** (respectively **method**) be denoted by $\tau_1$ (respectively $\tau_2$). Then, the fact that $\tau_1 \neq \tau_2$ means that the set of methods that can be placed after the dot in the partial call chain `allowing(any(Object.class)).` is not necessarily the same set of methods that can be placed after the dot in the partial call chain

```
allowing(any(Object.class)).method("get.*").
```

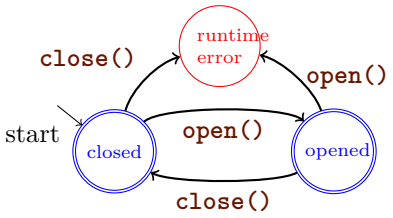
This distinction makes it possible to design expressive and rich fluent APIs, in which a sequence of "chained" calls is not only readable, but also robust, in the sense that the sequence is type correct only when it makes sense semantically.

There is a large body of research on *type-states* (See e.g., review articles such as [3, 9]). Informally, an object that belongs to a certain type, has type-states, if not all methods defined in this object's class are applicable to the object in all states it may be in. As it turns out, objects with type states are quite frequent: a recent study [8] estimates that about 7.2% of JAVA classes define protocols, that can be interpreted as type-state.

In a sense, type states define the "language" of the protocol of an object. The protocol of the type-state **Box** class defined in Figure 3 admits the chain `new Box().open().close()` but not the chain `new Box().open().open()`.

| | `open()` | `close()` |
|---|---|---|
| "closed" | *become "open"* | *runtime error* |
| "open" | *runtime error* | *become "closed"* |

**(a)** Definition by table




**(b)** Definition by DFA

**Figure 3** Fluent API of a box object, defined by a DFA and a table.

As mentioned above, tools such as fluflu realize type-state based on their finite automaton description. Our approach is a bit more expressive: examine the language $L$ defined by the type-state, e.g., in the box example,

$$L = \big(\texttt{.open().close()}\big)^*\big(\texttt{.open()} \mid \epsilon\big).$$

If $L$ is deterministic context-free, a fluent API can be made for it.

To make the proof concrete, consider this example of fluent API definition: An instance of class `Box` may receive two method invocations: `open()` and `close()`, and can be in either "open" or "closed" state. Initially the instance is "closed". Its behavior henceforth is defined by Figure 3.

To realize this definition, we need a type definition by which `new Box().open().close()`, more generally blue, or accepting states in the figure, type-check. Conversely, with this type definition, compile time type error should occur in `new Box().close()`, and, more generally, in the red state.

Some skill is required to make this type definition: proper design of class `Box`, perhaps with some auxiliary classes extending it, an appropriate method definition here and there, etc.

## 3 Context-Free Languages and Pushdown Automata: Reminder and Terminology

Notions discussed here are probably common knowledge (see e.g., [26, 32] for a text book description, or [6] for a scientific review). The purpose here is to set a unifying common vocabulary.

Let $\Sigma$ be a finite alphabet of *terminals* (often called input characters or tokens). A *language* over $\Sigma$ is a subset of $\Sigma^*$. Keep $\Sigma$ implicit henceforth.

A ***N****ondeterministic* ***P****ushdown* ***A****utomaton* (NPDA) is a device for language recognition, made of a nondeterministic finite automaton and a stack of unbounded depth of (stack) *elements*. A NPDA begins execution with a single copy of the initial element on the stack. In each step, the NPDA examines the next input token, the state of the automaton, and the top of the stack. It then pops the top element from the stack, and nondeterministically chooses which actions of its transition function to perform: Consuming the next input token, moving to a new state, or, pushing any number of elements to the stack. Actually, any combination of these actions may be selected.

The language recognized by a NPDA is the set of strings that it accepts, either by reaching an accepting state or by encountering an empty stack.

A ***C**ontext-**F**ree **G**rammar*(CFG) is a formal description of a language. A CFG $G$ has three components: $\Xi$ a set of *variables* (also called nonterminals), a unique *start variable* $\xi \in \Xi$, and a finite set of (production) *rules*. A rule $r \in G$ describes the derivation of a variable $\xi \in \Xi$ into a string of *symbols*, where symbols are either terminals or variables. Accordingly, rule $r \in G$ is written as $r = \xi \to \beta$, where $\beta \in (\Sigma \cup \Xi)^*$. This description is often called BNF. The *language* of a CFG is the set of strings of terminals (and terminals only) that can be derived from the start symbol, following any sequence of applications of the rules. CFG languages make a proper superset of regular languages, and a proper subset of "context-sensitive" languages [26].

The expressive power of NPDAs and BNFs is the same: For every language defined by a BNF, there exists a NPDA that recognizes it. Conversely, there is a BNF definition for any language recognized by some NPDA.

NPDAs run in exponential deterministic time. A more sane, but weaker, alternative is found in LR(1) parsers, which are deterministic linear time and space. Such parsers employ a stack and a finite automaton structure, to parse the input. More generally, LR($k$) parsers, $k > 1$, can be defined. These make their decisions based on the next $k$ input character, rather than just the first of these. General LR($k$) parsers are rarely used, since they offer essentially the same expressive power[11], at a greater toll on resources (e.g., size of the automaton). In fact, the expressive power of LR($k$), $k \geq 1$ parsers, is that of "***D**eterministic **P**ushdown **A**utomaton*" (DPDA), which are similar to NPDA, except that their conduct is deterministic.

▶ Definition 1 (Deterministic Pushdown Automaton). *A deterministic pushdown automaton (DPDA) is a quintuple* $\langle Q, \gamma, q_0, A, \delta \rangle$ *where* $Q$ *is a finite set of* states, $\gamma$ *is a finite* set of elements, $q_0 \in Q$ *is the initial state, and* $A \subseteq Q$ *is the* set of accepting states *while* $\delta$ *is the* partial state transition function $\delta : Q \times \gamma \times (\Sigma \cup \{\epsilon\}) \nrightarrow Q \times \gamma^*$.

*A DPDA begins its work in state* $q_0$ *with a single designated stack element residing on the stack. At each step, the automaton examines: the current state* $q \in Q$*, the element* $\gamma \in \gamma$ *at the top of the stack, and* $\sigma$*, the next input token, Based on the values of these, it decides how to proceed:*

1. *If* $q \in A$ *and the input is exhausted, the automaton accepts the input and stops.*
2. *Suppose that* $\delta(q, \gamma, \epsilon) \neq \bot$ *(in this case, the definition of a DPDA requires that* $\delta(q, \gamma, \sigma') = \bot$ *for all* $\sigma' \in \Sigma$*), and let* $\delta(q, \gamma, \epsilon) = (q', \zeta)$*. Then the automaton pops* $\gamma$ *and pushes the string of stack elements* $\zeta \in \gamma^*$ *into the stack.*
3. *If* $\delta(q, \gamma, \sigma) = (q', \zeta)$*, then the same happens, but the automaton also irrevocably consumes the token* $\sigma$*.*
4. *If* $\delta(q, \gamma, \epsilon) = \delta(q, \gamma, \sigma) = \bot$ *the automaton rejects the input and stops.*

A *configuration* is the pair of the current state and the stack contents. Configurations represent the complete information on the state of an automaton at any given point during its computation. A *transition* of a DPDA takes it from one configuration to another. Transitions which do not consume an input character are called *$\epsilon$-transitions*.

As mentioned above, NPDA languages are the same as CFG languages. Equivalently, *DCFG languages* (deterministic context-free grammar languages) are context-free languages that are recognizable by a DPDA. The set of DCFG languages is still a proper superset of regular languages, but a proper subset of CFG languages.

[11] they recognize the same set of languages [30].

## 4 Statement of the Main Result

Let java be a function that translates a terminal $\sigma \in \Sigma$ into a call to a uniquely named function (with respect to $\sigma$). Let java($\alpha$), be the function that translates a string $\alpha \in \Sigma^*$ into a fluent API call chain. If $\alpha = \sigma_1 \cdots \sigma_n \in \Sigma^*$, then

$$\mathsf{java}(\alpha) = \mathsf{java}(\sigma_1)\texttt{().}\cdots\texttt{.}\mathsf{java}(\sigma_n)\texttt{()}$$

For example, when $\Sigma = \{a, b, c\}$ let java($a$) = `a`, java($b$) = `b`, and, java($c$) = `c`. With these,

$$\mathsf{java}(caba) = \texttt{c().a().b().a()}$$

▶ **Theorem 1.** *Let $A$ be a DPDA recognizing a language $L \subseteq \Sigma^*$. Then, there exists a* Java *type definition, $J_A$ for types* `L`, `A` *and other types such that the* Java *command*

$$\texttt{L } \ell \texttt{ = A.build.}\mathbf{\mathsf{java}}(\alpha)\texttt{.\$();} \tag{1}$$

*type checks against $J_A$ if an only if $\alpha \in L$. Furthermore, program $J_A$ can be effectively generated from $A$.*

Equation 1 reads: starting from the `static` field `build` of `class` `A`, apply the sequence of call chain java($\alpha$), terminate with a call to the ending character `$()` and then assign to newly declared Java variable $\ell$ of type `L`.

The proof of the theorem is by a scheme for encoding in Java types the pushdown automaton $A = A(L)$ that recognizes language $L$. Concretely, the scheme assigns a type $\tau(c)$ to each possible configuration $c$ of $A$. Also, the type of `A.build` is $\tau(c_0)$, where $c_0$ is the initial configuration of $A$,

Further, in each such type the scheme places a function $\sigma()$ for every $\sigma \in \Sigma$. Suppose that $A$ takes a transition from configuration $c_i$ to configuration $c_j$ in response to an input character $\sigma_k$. Then, the return type of function $\sigma_k$`()` in type $\tau(c_i)$ is type $\tau(c_j)$.

With this encoding the call chain in Equation 1 mimics the computation of $A$, starting at $c_0$ and ending with rejection or acceptance. The full proof is in Section 7.

Since the depth of the stack is unbounded, the number of configurations of $A$ is unbounded, and the scheme must generate an infinite number of types. Genericity makes this possible, since a generic type is actually device for creating an unbounded number of types.

There are several, mostly minor, differences between the structure of the Java code in Equation 1 and the examples of fluent API we saw above,e.g., in Figure 1:

**Prefix, i.e., the starting `A.build` variable.** All variables and functions of Java are defined within a class. Therefore, a call chain must start with an object (`A.build` in Equation 1) or, in case of `static` methods, with the name of a class. In fluent API frameworks this prefix is typically eliminated with appropriate `import` statements.

If so desired, the same can be done by our type encoding scheme: define all methods in type $\tau(c_0)$ as `static` and `import static` these.

**Suffix, i.e., the terminal `.$()` call.** In order to know whether $\alpha \in L$ the automaton recognizing $L$ must know when $\alpha$ is terminated.

With a bit of engineering, this suffix can also be eliminated. One way of doing so is by defining type `L` as an `interface`, and by making all types $\tau(c)$, $c$ is an accepting configuration, as subtype of `L`.

**Parameterized methods.** Fluent API frameworks support call chains with phrases such as:

- "`.when(header(foo).isEqualTo("bar")).`",
- "`.and(BOOK.PUBLISHED.gt(date("2008-01-01"))).`", and,

- "`.allowing(any(Object.class)).`".

while our encoding scheme assumes methods with no parameters.

Methods with parameters contribute to the user experience and readability of fluent APIs but their "computational expressive power"' is the same. In fact, extending Theorem 1 to support these requires these conceptually simple steps

1. Define the structure of parameters to methods with appropriate fluent API, which may or may not be, the same as the fluent API of the outer chain, or the fluent API of parameters to other methods. Apply the theorem to each of these fluent APIs.
2. If there are several overloaded versions of a method, consider each such version as a distinct character in the alphabet $\Sigma$ and in the type encoding of the automaton.
3. Add code to the implementation of each method code to store the value of its argument(s) in a record placed at the end of the fluent-call-list.

## 5 Techniques of Type Encoding

This section presents techniques and idioms of type encoding in JAVA partly to serve in the proof of Theorem 1, and partly to familiarize the reader with the challenges of type encoding.

Let $g : \gamma \nrightarrow \gamma$ be a partial function, from the finite set $\gamma$ into itself. We argue that $g$ can be represented using the compile-time mechanism of JAVA. Figure 4 encodes such a partial function for $\gamma = \{\gamma_1, \gamma_2\}$, where $g(\gamma_1) = \gamma_2$ and $g(\gamma_2) = \perp$, i.e., $g(\gamma_2)$ is undefined.[12]

The type hierarchy depicted in Figure 4(a) shows five classes: Abstract class $\gamma$ [13] represents the set $\gamma$, final classes $\gamma$`1`, $\gamma$`2` that extend $\gamma$, represent the actual members of the set $\gamma$. The remaining two classes are private final class ¤ that stands for an error value, and abstract class $\gamma'$ that denotes the augmented set $\gamma \cup \{¤\}$. Accordingly, both classes ¤ and $\gamma$ extend $\gamma'$.[14]

The full implementation of these classes is provided in Figure 4(b). This actual code excerpt should be placed as a nested class of some appropriate host class. Import statements are omitted, here and henceforth for brevity.

The use cases in Figure 4(c) explain better what we mean in saying that function $g$ is encoded in the type system: An instance of class $\gamma$`1` returns a value of type $\gamma$`2` upon method call `g()`, while an instance of class $\gamma$`2` returns a value of our `private` error type $\gamma'$`.¤` upon the same call.

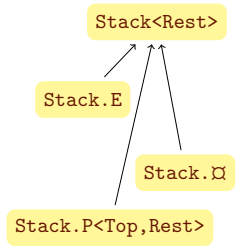
Three recurring idioms employed in Figure 4(b) are:

1. An `abstract` class encodes a set (alternatively, one can use `interfaces`). Abstract classes that extend it encode subsets, while `final` classes encode set members.
2. The interest of frugal management of name-spaces is served by the agreement that if a class $X$ `extends` another class $Y$, then $X$ is also defined as a `static` member class of $Y$.
3. Bodies of functions are limited to a single `return null;` command (with interfaces the method body is redundant). This is to stress that at runtime, the code does not carry

[12] Unless otherwise stated, all code excerpts here represent full implementations, and automatically extracted, omitting headers and footers, from JAVA programs that compile correctly with a JAVA 8 compiler.

[13] Remember that JAVA admits Unicode characters in identifier names

[14] The use of short names, e.g., $\gamma$ instead of $\gamma'.\gamma$, is made possible by an appropriate `import` statement omitted here and henceforth.




**(a)** type hierarchy

```
public static abstract class Γ′ {
  private static abstract class ¤ extends Γ′ {
    // Empty private class, cannot be used by clients.
    private ¤() { /* Private constructor hinders extension by clients */
      }
  }
  public  static abstract class γ extends γ′ {
    public abstract γ′ g();
    public static final class γ1 extends γ {
      // Covariant return type in overriding:
      @Override public γ2 g() { return null; }
    }
    public static final class γ2 extends γ {
      // Covariant return type in overriding:
      @Override  public γ′.¤ g() { return null; }
    }
  }
}
```

**(b)** implementation

```
public static void five_use_cases_of_function_g() {
  γ2 _1 = new γ1().g(); // ✓
  γ1 _2 = new γ2().g(); // ✗ type mismatch
  γ′.¤ _3 = new γ2().g(); // ✗ class ¤ is private
  γ′ _4 = new γ2().g(); // ✓
  _4.g(); // ✗ g() undefined in type γ′
}
```

**(c)** use cases

**Figure 4** Type encoding of partial function $g : \gamma \nrightarrow \gamma$, defined by $\gamma = \{\gamma_1, \gamma_2\}$, $g(\gamma_1) = \gamma_2$ and $g(\gamma_2) = \bot$.

out any useful or interesting computation, and the class structure is solely for providing compile-time type checks. [15]

Having seen how inheritance and overriding make possible the encoding of unary functions, we turn now to encoding higher arity functions. With the absence of multi-methods, other techniques must be used.

Consider the partial binary function $f : R \times S \nrightarrow \gamma$, defined by

$$\begin{array}{lll} R = \{r_1, r_2\} & f(r_1, s_1) = \gamma_1 & f(r_2, s_1) = \gamma_1 \\ S = \{s_1, s_2\} & f(r_1, s_2) = \gamma_2 & f(r_2, s_2) = \bot \end{array} . \tag{2}$$

A Java type encoding of this definition of function $f$ is in Figure 5(a); use cases are in Figure 5(b).

As the figure shows, to compute $f(r_1, s_1)$ at compile time we write `f.r1().s1()`. Also, the fluent API call chain `f.r2().s2().g()` results in a compile time error because

$f(r_2, s_2) = \bot$.

Class `f` in the implementation sub-figure serves as the starting point of the little fluent API defined here. The return type of `static` member functions `r1()` and `r2()` is the respective sub-class of class `R`: The return type of function `r1()` is class `R.r1`; the return type of function `r2()` is class `R.r2`.

[15] A consequence of these idioms is that the augmented class $\gamma'$ is visible to clients. It can be made `private`. Just move class $\gamma$ to outside of $\gamma'$, defying the second idiom.

```
public static abstract class f { // Starting point of fluent API
  public static r1 r1() { return null; }
  public static r2 r2() { return null; }
}
public static abstract class R {
  public abstract γ′ s1();
  public abstract γ′ s2();
  public static final class r1 extends R {
    @Override public γ1 s1() { return null; }
    @Override public γ2 s2() { return null; }
  }
  public static final class r2 extends R {
    @Override public γ2 s1() { return null; }
    @Override public γ′.¤ s2() { return null; }
  }
}
```

**(a)** implementation (except for classes $\gamma$, $\gamma'$, $\gamma 1$, and $\gamma 2$, found in Figure 4).

```
public static void four_use_cases_of_function_f() {
  γ1 _1 = f.r1().s1(); // ✓ f(r1, s1) = γ1
  γ2 _2 = f.r1().s2(); // ✓ f(r1, s2) = γ2
  γ2 _3 = f.r2().s1(); // ✓ f(r2, s1) = γ2
  f.r2().s2().g(); // ✗ method s2() undefined in type Γ'
}
```

**(b)** use cases

**Figure 5** Type encoding of partial binary function $f : R \times S \nrightarrow \gamma$, where $R = \{r_1, r_2\}$, $S = \{s_1, s_2\}$, and $f$ is specified by $f(r_1, s_1) = \gamma_1$, $f(r_1, s_2) = \gamma_2$, $f(r_2, s_1) = \gamma_1$, and $f(r_2, s_2) = \bot$.

```
interface ID<T extends ID<?>> {
  default T id() { return null; }
}
class A implements ID<A> { /**/ }
abstract class B<Z extends B<?>> implements ID<Z> { /**/ }
class C extends B<C> { /**/ }
```

**Figure 6** Covariant return type of function `id()` with Java generics.

Instead of representing set $S$ as a class, its members are realized as methods `s1()` and `s2()` in class `R`. These functions are defined as `abstract` with return type `γ'` in `R`. Both functions are overridden in classes `r1` and `r2`, with the appropriate covariant change of their return type,

It should be clear now that the encoding scheme presented in Figure 5 can be generalized to functions with any number of arguments, provided that the domain and range sets are finite. The encoding of sets of unbounded size require means for creating an unbounded number of types. Genericity can be employed to serve this end.

Figure 6 shows a genericity based recipe for a function whose return type is the same as the receiver's type. This recipe is applied in the figure to classes `A`, `B`, and `C`. In each of these classes, the return type of `id` is, without overriding, (at least) the class itself.

It is also possible to encode with Java generic types unbounded data structures, as demonstrated in Figure 7, featuring a use case of a stack of an *unbounded* depth.

In line 3 of the figure, a stack with five elements is created: These are popped in order (ll.4–7,l.9). Just before popping the last item, its value is examined (l.8). Trying then to pop from an empty stack (l.10), or to examine its top (l.11), ends with a compile time error.

Stack elements may be drawn from some abstract set $\gamma$. In the figure these are either class $\gamma 1$ or class $\gamma 2$ (both defined in Figure 4). A call to function $\gamma i$ pushes the type $\gamma i$ into

```
1 public static void use_case_of_stack() {
2   // Create a stack a with five items in it:
3   P<γ1, P<γ1, P<γ2, P<γ1, P<γ1, E>>>>> _1 = Stack.empty.γ1().γ1().γ2().γ1().γ1();
4   P<γ1, P<γ2, P<γ1, P<γ1, E>>>> _2 = _1.pop(); // ✓ Pop one item
5   P<γ2, P<γ1, P<γ1, E>>> _3 = _2.pop(); // ✓ Pop another item
6   P<γ1, P<γ1, E>> _4 = _3.pop(); // ✓ Pop yet another item
7   P<γ1, E> _5 = _4.pop(); // ✓ Pop penultimate item
8   γ1 _6 = _5.top(); // ✓ Examine last item
9   E _7 = _5.pop(); // ✓ Pop last item
10  Stack.¤ _8 = _7.pop(); // ✗ Cannot pop from an empty stack
11  γ′.¤ _9 = _7.top(); // ✗ empty stack has no top element
12 }
```

**Figure 7** Use cases of a compile-time stack data structure.

the stack, for $i = 1,$. The expression

`Stack.empty.γ1().γ1().γ2().γ1().γ1()`

represents the sequence of pushing the value $\gamma_1$ into an empty stack, followed by $\gamma_1$, $\gamma_2$, $\gamma_1$, and, finally, $\gamma_1$. This expression's type is that of variable `_1`, i.e.,

`P<γ1,P<γ1,P<γ2,P<γ1,P<γ1,E»»>`

A recurring building block occurs in this type: generic type `P`, *short for "Push"*, which takes two parameters:

1. the *top* of the stack, always a subtype of $\gamma$,
2. the *rest* of the stack, which can be of two kinds:
   a. another instantiation of `P` (in most cases),
   b. non-generic type `E`, *short for "Empty"*, which encodes the empty stack. Note that `E` can only occur at the deepest `P`, encoding a stack with one element, in which the rest is empty.

Incidentally, `static` field `Stack.empty` is of type `E`.

Figure 8(a) gives the type inheritance hierarchy of type `Stack` and its subtypes. Figure 8(b) gives the implementation of these types.

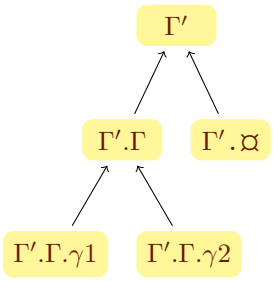
The code in the figure shows that the "rest" parameter of `P` must extend class `Stack`, and that both types `P` and `E` extend `Stack`. Other points to notice are:

- The type at the top of the stack is precisely the return type of `top()`; it is overridden in `P` so that its return type is the first argument of `P`. The return type of `top()` in `E` is the error value $\gamma'.¤$.
- Pushing into the stack is encoded as functions `γ1()` and `γ2()`; the two are overridden with appropriate covariant change of the return type in `P` and `E`.
- Since an empty stack cannot be popped, function `pop()` is overridden in `E` to return the *error* type `Stack.¤`. This type is indeed a kind of a stack, except that each of the four stack functions: `top()`, `push()`, `γ1()`, and, `γ2()`, return an appropriate error type.

In fact, this recursive generic type technique can used to encode S-expressions: In the spirit of Figure 8, the idea is to make use of a `Cons` generic type with covariant `car()` and `cdr()` methods.

A standard technique of template programming in C++ is to encode conditionals with template specialization. Since Java forbids specialization of generics, in lieu we use covariant overloading of function return type (e.g., the return type of `s2()` in Figure 5 and the return type of `top()` in Figure 8).

Figure 9 shows that a similar covariant change is possible in extending a generic type. The type of the parameter `M` to `Heap` is "`? extends Mammals`". This type is specialized as




**(a)** type hierarchy

```
public static abstract class Stack<Rest extends Stack<?>> {
  public abstract γ′ top();
  public abstract Rest pop();
  public abstract Stack<?> γ1();   // Push type Γ′.Γ.γ1
  public abstract Stack<?> γ2();   // Push type Γ′.Γ.γ2
  public static class P<Top extends γ, Rest extends Stack<?>>
    extends Stack<Rest> { // Type of a non−empty stack:
      @Override public Top top() { return null; }
      @Override public Rest pop() { return null; }
      @Override public P<γ1, P<Top,Rest>> γ1() { return null; }
      @Override public P<γ2, P<Top,Rest>> γ2() { return null; }
  }
  public static final class E
    extends Stack<¤> { // Type of an empty stack
      @Override public γ′.¤ top() { return null; }
      @Override public ¤ pop() { return null; }
      @Override public P<γ1, E> γ1() { return null; }
      @Override public P<γ2, E> γ2() { return null; }
  }
  public static final E empty = null;
  private static final class ¤
    extends Stack<¤> { // Type of pop from empty stack
      @Override public γ′.¤ top() { return null; }
      @Override public ¤ pop() { return null; }
      @Override public ¤ γ1() { return null; }
      @Override public ¤ γ2() { return null; }
  }
}
```

**(b)** implementation (except for classes $\gamma$, $\gamma'$, $\gamma 1$, and $\gamma 2$, from Figure 4).

**Figure 8** Type encoding of an unbounded stack data structure.

```
class Mammals { /* ... */ }
class Heap<M extends Mammals> { /* ... */}
class Whales extends Mammals { /* ... */}
class School<W extends Whales>
  extends Heap<W> { /* ... */}
```

**Figure 9** Covariance of parameters to generics.

`School` extends `Heap`: parameter `W` of `School` is of type "`? extends Whales`". Covariant specialization of parameters to generics is yet another idiom for encoding conditionals.

Overloading gives rise to a third idiom for partial emulation of conditionals, as can be seen in Figure 10.

The figure depicts type `Peep` and overloaded versions of `peep()` which together make it possible to extract the top of the stack. The first generic parameter to `Peep` is the top of the stack, the second is the stack itself. Indeed, we see (l.11) that peeping into an empty stack, places a `?` in the first parameter, thanks to the first overloaded version of `peep()` (l.2).

The second overloaded version of `peep()` (ll.3–6) matches against all non-empty stacks. The return type of this version encodes in its first parameter the top of the stack, and in its second parameter, the parameter's type. A use case is in line 9.

Let $\tau$ be the type of the top of a given stack. Then, both `top()` and `peep()` can be used to extract $\tau$. There is a subtle difference between the two though: Obtaining $\tau$ from `top()` does not make it possible to define variables, function return types, and parameters to functions and generics whose type is $\tau$ or depends on it in any way. However, since `Peep` is a type that receives $\tau$ as parameter, the body of `Peep` is free to define e.g., functions signature includes on $\tau$, or pass $\tau$ further to other generics.

```
1 public static class Peep<γ extends γ′, S extends Stack<? extends Stack<?>>> {}
2 public static Peep<?, E> peep(E _) { return null; } // First overloaded version of peep()
3 public static                                        // Second overloaded version of peep()
4   <Top extends γ, Rest extends Stack<?>> // Two generic parameters
5   Peep<Top, P<Top, Rest>>                // Function return type
6   peep(P<Top, Rest> _) { return null; }  // Function parameters and body
7 public static void peeping_into_a_stack_use_cases() {
8   P<γ2, P<γ1, P<γ2, P<γ1, P<γ2, E>>>>> _1 = Stack.empty.γ2().γ1().γ2().γ1().γ2();
9   Peep<γ2, P<γ2, P<γ1, P<γ2, P<γ1, P<γ2, E>>>>>> _2 = peep(_1);
10  E _3 = Stack.empty;
11  Peep<?, E> _4 = peep(_3);
12 }
```

**Figure 10** Peeping into the stack.

```
1 public interface JS<  // 1 + k generic parameters
2     Rest extends JS,              // As a convention, we use JS
3     J_γ1 extends JS,     // with its raw type when no
4     J_γ2 extends JS      // parameters are introduced
5 > {
6   γ′ top();
7   Rest pop();
8   JS γ1();
9   JS γ2();
10  J_γ1 jump_γ1();
11  J_γ2 jump_γ2();
12  interface ¤ extends JS<¤, ¤, ¤> { ... }
13  public interface E extends JS<¤, ¤, ¤> { ... }
14  public static final E empty = null;
15  public interface P<// 2 + k generic arguments:
16      Top extends γ,
17      Rest extends JS,
18      J_γ1 extends JS,
19      J_γ2 extends JS
20  > extends P′<Top, Rest, J_γ1, J_γ2,
21        P<Top, Rest, J_γ1, J_γ2>
22  > { /**/ }
23 }
```

**Figure 11** Skeleton of type encoding for the jump-stack data structure.

## 6 The Jump-Stack Data-Structure

A *jump-stack* is a stack data structure whose elements are drawn from a finite set $\gamma$, except that jump-stack supports $\mathsf{jump}(\gamma)$, $\gamma \in \Gamma$ operations (which means "repetitively *pop* elements from the stack up to and including the first occurrence of $\gamma$").

Figure 11 shows the skeleton of type-encoding, in parameterized type `JS`, of a jump-stack whose elements are drawn from type $\gamma$ (Figure 4(b)), i.e., either $\gamma$`1` or $\gamma$`2`.

Just like `Stack` (Figure 8(b)), `JS` takes a `Rest` parameter encoding the type of a jump-stack after popping. In addition `JS` takes $k = |\gamma|$ type parameters, one for each $\gamma \in \gamma$, which is the type encoding of the jump-stack after a $\mathsf{jump}(\gamma)$ operation. In the figure, there are two such parameters: `J_`$\gamma$`1`, and `J_`$\gamma$`2`.

Functions defined in `JS` include not only the standard stack operations: `top()`, `pop()`, $\gamma$`1()` and $\gamma$`2()` (encoding a push of $\gamma_i$, $i = 1, 2$, in general, there are $k$), but also $k$ functions encoding $\mathsf{jump}(\gamma)$, $\gamma \in \gamma$. In our case, these are `jump_`$\gamma$`1` and `jump_`$\gamma$`2`, which encode $\mathsf{jump}(\gamma_i)$ thanks to their return type being `J_`$\gamma i$, $i = 1, 2$.

The type hierarchy rooted at `JS` is similar to that of Figure 8(a): Two of the specializations are parameter-less and are almost identical to their `Stack` counterparts: `JS.E` encodes an empty jump-stack; `JS.¤` encodes a jump-stack in error, e.g., after popping from `JS.E`. The body of these two types is omitted here.

Type `JS.P` (lines 16–23 in Figure 11) makes the third specialization of `JS`, encoding a stack with one or more elements. Just like in Figure 6, there are no overridden functions in

```
1 private interface P'<
2 // 2 + k + 1 generic arguments:
3   Top extends γ,
4   Rest extends JS,
5   J_γ1 extends JS,
6   J_γ2 extends JS,
7   Me extends JS
8 > extends JS<Rest, J_γ1, J_γ2> {
9    public Top top();
10   P<γ1, Me, Me, J_γ2> γ1();
11   P<γ2, Me, J_γ2, Me> γ2();
12 }
```

**Figure 12** Auxiliary type `P'` encoding succinctly a non-empty jump-stack.

```
public static void jump_stack_use_cases(){
  P<
    γ1,                                  // Top
    P<γ1,P<γ2,E,¤>,P<γ2,E,¤>,E>,E>,// Rest
    P<γ1,P<γ2,E,¤>,P<γ2,E,¤>,E>,E>,// jump(γ1)
    E                                    // jump(γ2)
  > _1 = JS.empty.γ2().γ1().γ1();
  E _2 = _1.jump_γ2();
  P<
    γ1,              // Top
    P<γ2,E,¤>,E>,    // Rest
    P<γ2,E,¤>,E>     // jump(γ1)
    ,E               // jump(γ2)
  > _3 = _1.jump_γ1();
}
```

**Figure 13** Use cases for the `JS` type hierarchy.

`JS.P`; it fulfills its duties through the type parameters it takes and the types it passes to `P'` the generic type it extends.

Specifically, `JS.P` takes the same `Top` and `Rest` parameters (ll.17–18) as type `Stack.P`: as well as $k$ additional parameters: `J_γ1` and `J_γ2` (ll.19–20) which are the types encoding the jump-stack after the execution $\mathsf{jump}(\gamma_i)$, $i = 1, 2$. Type `JP.P` passes these four parameters to type `P'` which it extends (l.21). The fifth parameter to `P'` (l.22) is the current incarnation of `P`, i.e., `P<Top, Rest, J_γ1, J_γ2`.

The auxiliary (and `private`) type `P'` itself is depicted in Figure 12. By extending type `JS` and passing the correct `Rest` (respectively, `J_γ1`, `J_γ2`) parameter to it, `P'` inherits correct declaration of function `pop()` (l.8 Figure 11) (respectively `jump_γ1` (l.11 ibid), `jump_γ2` (l.12 ibid)).

More importantly, the `Me` type parameter to `P'` represents type `JP.P` that extends `P'`. Type `Me` also captures the actual parameters *included* to `JP.P`, which makes it possible to write the return type of `γ1()` and `γ2()` more succinctly. Let, e.g., $\tau$ = `P<γ1, Me, Me, J_γ2>` be the return type of `γ1()`. The first two parameters to $\tau$ say that pushing `γ1`, results in a compound jump-stack, whose top element is `γ1`, and where the rest of the jump-stack is the current type. The third parameter to $\tau$ says that since `γ1` was pushed the result of a $\mathsf{jump}(\gamma_1)$ is the type of the receiver. The fourth parameter is `J_γ2` since a push of $\gamma_1$ does not change the result of $\mathsf{jump}(\gamma_2)$.

Some use cases for the encoded jump-stack data structure are in Figure 13. The type of variable `_1` encodes a stack into which `γ2`, `γ1`, `γ1` were pushed (in this order). Examining the type of `_2` we see that executing `jump_γ2` on `_1`, yields the empty stack in a single step. The type of `_3` is that state of the same stack after executing `jump_γ1`; it is exactly the same as popping a single element from the stack.

## 7 Proof of Theorem 1

On a first sight, the proof of Theorem 1 could follow the techniques sketched in Section 5 to type encode a DPDA (Definition 1). The partial transition function $\delta$ may be type encoded as in Figure 5(a), and the stack data structure of a DPDA can be encoded as in Figure 8.

The techniques however fail with $\epsilon$-transitions, which allow the automaton to move between an unbounded number of configurations and maneuver the stack in a non-trivial manner, without making any progress on the input. The fault in the scheme lies with compile time computation being carried out by the $\mathsf{java}(\sigma)()$ functions, each converting their receiver type to the type of the receiver of the next call in the chain. We are not aware of a JAVA type encoding which makes it possible to convert an input type into an output type, where the output is computed from the input by an unbounded number of steps. [16]

The literature speaks of finite-delay DPDAs, in which the number of consecutive $\epsilon$-transitions is uniformly bounded and even of realtime DPDAs in which this bound is 0, i.e., no $\epsilon$-transitions. Our proof relies on a special kind of realtime automata, described by Courcelle [14].

▶ Definition 2 (Simple-Jump Single-State Realtime Deterministic Pushdown Automaton). *A simple-jump, single-state, realtime deterministic pushdown automaton (jDPDA, for short) is a triplet* $\langle \gamma, \gamma_1, \delta \rangle$ *where* $\gamma$ *is a set of stack elements,* $\gamma_1 \in \gamma$ *is the initial stack element, and* $\delta$ *is the* partial transition function, $\delta : \gamma \times \Sigma \nrightarrow \gamma^* \cup j(\gamma)$,

$$j(\gamma) = \{\textit{instruction } \mathsf{jump}(\gamma) \mid \gamma \in \gamma\}.$$

*A configuration of a jDPDA is some* $c \in \gamma^*$ *representing the stack contents. Initially, the stack holds* $\gamma_1$ *only. For technical reasons, assume that the input terminates with* $\$ \notin \Sigma$*, a special end-of-file character.*

- *At each step a jDPDA examines* $\gamma$*, the element at the top of the stack, and* $\sigma \in \Sigma$*, the next input character, and executes the following:*
  1. *consume* $\sigma$
  2. *if* $\delta(\gamma, \sigma) = \zeta$, $\zeta \in \gamma^*$*, the automaton pops* $\gamma$*, and pushes* $\zeta$ *into the stack.*
  3. *if* $\delta(\gamma, \sigma) = \mathit{jump}(\gamma')$, $\gamma' \in \gamma$*, then the automaton repetitively pops stack elements up-to and including the first occurrence of* $\gamma'$*.*
- *If the next character is* \$*, the automaton may reject or accept (but nothing else), depending on the value of* $\gamma$*.*

*In addition, the automaton rejects if* $\delta(\gamma, \sigma) = \bot$ *(i.e., undefined), or if it encounters an empty stack (either at the beginning of a step or on course of a jump operation).*

As it turns out, every DCFG language is recognized by some jDPDA, and conversely, every language accepted by a jDPDA is a DCFG language [14]. The proof of Theorem 1 is therefore reduced to type-encoding of a given jDPDA. Towards this end, we employ the type-encoding techniques developed above, and, in particular, the jump-stack data structure (Figure 11).

[16] With the presumption that the JAVA compiler halts for all inputs (a presumption that does not hold for e.g., C++, and was never proved for JAVA), the claim that there is no JAVA type encoding for all DPDAs can be proved: Employing $\epsilon$-transitions, it is easy to construct an automaton $A^\infty$ that never halts on any input. A type encoding of $A^\infty$ creates programs that send the compiler in an infinite loop.

**Table 1** The transition function of a jDPDA $A$, $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$, $\gamma = \{\gamma_1, \gamma_2\}$ where $\gamma_1$ is the initial element.

| | $\gamma_1$ | $\gamma_2$ |
|---|---|---|
| $\sigma_1$ | $\mathsf{push}(\gamma_1, \gamma_1, \gamma_2)$ | $\mathsf{push}(\gamma_2, \gamma_2)$ |
| $\sigma_2$ | $\perp$ | $\mathsf{push}(\epsilon)$ |
| $\sigma_3$ | $\perp$ | $\mathsf{jump}(\gamma_1)$ |
| \$ | accept | reject |

Henceforth, let $k = |\gamma|$, $\ell = |\Sigma|$. The simple $k = 2$, $\ell = 3$ jDPDA $A$ defined in Table 1 will serve as our running example. Let $L$ be the language recognized by $A$.[17]

## 7.1 Main Types

Generation of a type encoding for a jDPDA starts with two empty types for sets $L$, $\Sigma^*$, where $L$ represents the languages accepted by the jDPDA and $\Sigma^*$ represents all words:

```
private static class ΣΣ    // Encodes set Σ*, type of reject
  { /* empty */  }
static class L extends ΣΣ // Encodes set L ⊆ Σ*, type of accept
  { /* empty */  }
```

(The full type encoding is in Figure 15 below; to streamline the reading, we bring excerpts as necessary.)

A configuration is encoded by a generic type `C`. Essentially, `C` is a representation of the stack, but $k+1$ type parameters are required:

- `Rest`, a type encoding of the stack after a pop (or jump with the top element), and,
- $k$ types, named `JR`$\gamma$`1`, ... ,`JR`$\gamma k$, encoding the type of `Rest` after $\mathsf{jump}(\gamma_1)$,... $\mathsf{jump}(\gamma_k)$.

Note that these $k+1$ parameters are sufficient for describing a configuration, i.e., if the top is $\gamma_j$, then for all $j \neq i$

$$\mathsf{jump}(\gamma_j) = \texttt{Rest.}\mathsf{jump}(\gamma_j)$$

In the special case of $\mathsf{jump}(\gamma_i)$ the returned type is still `Rest`, this is due to the fact that before a jump operation, we do not pop an element from the stack.

All instantiations of `C` must make sure that actual parameters are properly constrained, to ensure that they are (the type version of) pointers into the actual stack, not a trivial task, as will be seen shortly.

[17] Incidentally,

$$L = \left\{ w^* \mid w = (\sigma_1^n \sigma_2^m \sigma_3 | \sigma_1^n \sigma_2^n) \ , \ n > m, n > 1 \right\}$$

which is clearly not-regular; the equivalent BNF for $L$ is:

$$S \to WS \,|\, \epsilon \ ; \ W \to D \,|\, AD\sigma_3 \ ; \ D \to \sigma_1 D \sigma_2 \,|\, \sigma_1\sigma_2 \ ; \ A \to A\sigma_1 \,|\, \sigma_1$$

Neither the BNF nor the representation are material for the proof.

```
static void isL(L l) {/**/}
static void accepts() {
  isL(A.build.$());
  isL(A.build.σ1().σ3().$());
  isL(A.build.σ1().σ2().$());
  isL(A.build.σ1().σ1().σ2().σ3().σ1().σ2().$());
}
static void rejects() {
  isL(A.build.σ1().$());
  isL(A.build.σ2().σ1().$());
  isL(A.build.σ1().σ2().σ3().$());
  isL(A.build.σ1().σ1().σ2().σ3().σ1().$());
}
```

**Figure 14** Accepting and non-accepting call chains with the type encoding of jDPDA $A$ (as defined in Table 1). All lines in `accepts()` type-check, while all lines in `rejects()` do not type-check.

In the running example, `C` is defined as:

```
interface C<       // Generic parameters:
  Rest extends C, // The rest of the stack, for pop or jump(γ) operations
  JRγ1 extends C, // Type of  Rest.jump(γ1), may be rest, or anything in it.
  JRγ2 extends C  // Type of  Rest.jump(γ2), may be rest, or anything in it.
>
{
  ΣΣ $();         // δ transition on end of input; invalid language by default
  C σ1();         // δ transition on σ1; dead end by default
  C σ2();         // δ transition on σ2; dead end by default
  C σ3();         // δ transition on σ3; dead end by default
  public interface E extends C<¤,¤,¤> { /* Empty stack configuration */ }
  interface ¤ extends C<¤,¤,¤> { /* Error configuration. */ }
}
```

This excerpt shows also classes `E` and `¤` which encode (as in Figure 11) the empty and the error configurations.

Type `C` defines $\ell + 1$ functions (4 in the example), one for each possible input character, and one for the end-of-file character defined as \$. Since `C` encodes an abstract configuration, return types of functions in it are the appropriate defaults which intentionally fail to emulate the automaton's execution. The return type of `$()` is $\Sigma\Sigma$ (rejection); the transition functions `σ1()`, ... `σℓ()`, return the raw type `C`.

## 7.2 Top-of-Stack Types

Types `Cγ1`, ... ,`Cγk`, specializing `C`, encode stacks whose top element is $\gamma_1$, ... ,$\gamma_k$. In $A$ there are two of these:

```
interface Cγ1< // Configuration when γ1 is at top
  Rest extends C, JRγ1 extends C, JRγ2 extends C
> extends
  C<Rest, JRγ1, JRγ2>
{
}
interface Cγ2< // Configuration when γ2 is at top
  Rest extends C, JRγ1 extends C, JRγ2 extends C
> extends
  C<Rest, JRγ1, JRγ2>
{
}
static Cγ1<E,¤,¤> build = null;
```

In $A$, types `Cγ1` and `Cγ2` take three parameters; in general "Top of Stack" types take the aforementioned $k + 1$ parameters.

The method signatures of these types are generated using the mentioned parameters. The generating of methods will be discussed next.

The code defines the `static` variable `build`, the starting point of all fluent API call chains, to be of type `C`$\gamma$`1<E,¤,¤>`, i.e., the starting configuration of the automaton is a stack whose top is $\gamma_1$, and its `Rest` parameter is empty (`E`). Any of the two jumps possible on this rest results with, `¤`, an undefined stack. Examples of accepting and rejecting call chains starting at `A.build` can be seen in Figure 14.

## 7.3 Transitions

It remains to show the type encoding of $\delta$, the transition function. Overall, there are a total of $k \cdot (\ell + 1)$ entries in a transition table such as Table 1. Conceptually, these are encoded by selecting the correct return type of functions $\sigma$`1()`, ... ,$\sigma k$`()` and `$()` in each of the $k$ "Top of Stack" types. Thanks to inheritance, we need to do so only in the cases that this return type is different from the default.

Overall, there are six kinds of entries in a transition table:

**reject** The default return type of `$()` in `C` is $\Sigma\Sigma$, which is *not* a subtype of `L`. Normally the result of a call chain that ends with `$()` cannot be assigned to a variable of type `L`. Moreover, since $\Sigma\Sigma$ is `private`, there is little that clients can do with this result.

**accept** The only case in which fluent call chain ending with `$()` can return type `L` is when the type returned of the call just prior to `.$()` covariantly changes the return type of `$()` to `L`.[18]

Recall that a jDPDA can only accept after its input is exhausted. In Table 1 we see that accept occurs when the top of the stack is $\gamma_1$. We therefore add to the body of type `C`$\gamma$`1` the line

```
@Override L $();
```

$\perp$ When a prefix of the input is sufficient to conclude it must be rejected however it continues, the transition function returns $\perp$. In $A$ this occurs when the top of the stack is $\gamma_1$ and one of $\sigma_2$ or $\sigma_3$ is read. To type encode $\delta(\gamma_1, \sigma_2) = \perp$, one must *not* override $\sigma$`2()` in type `C`$\gamma$`1`; the inherited return type (l.15 Figure 15) is the raw `C`. Subsequent calls in the chain will all receive and return a raw `C` (Recall that all $\sigma i$`()`, $i = 1, \ldots, \ell$, are functions in `C` that return a raw `C`). Therefore, the final `$()` will reject.

Two other situations in which a jDPDA rejects but not demonstrated in $A$ are: a jump that encounters an empty stack, and reading a character from when the stack is empty. In our type encoding these are handled by the special types `E` and `¤` (ll.17–18 ibid), both extend `C` without overriding any of its methods. Again, remaining part of the call chain will stick to raw `C`s up until the final `$()` call rejects the input.

**jump**($\gamma_i$) The design of the generic parameters makes the implementation of jump($\gamma_i$) operations particularly simple. All that is required is to covariantly change the return type of the appropriate $\sigma j$`()` function to the appropriate `JR`$\gamma i$ or `Rest` parameter (recall that a jump occurs after popping the current element from the stack, so we refer to `JR` type parameters rather than `J`'s).

In Table 1 we find that $\delta(\gamma_2, \sigma_3) =$ jump($\gamma_1$). Accordingly, the type of $\sigma$`2()` in `C`$\gamma$`2` (l.34) is `JR`$\gamma$`1`.

[18] This is not to be confused with dynamic binding; types of fluent API call chains are determined statically.

**push**($\zeta$) Push operations are the most complex, since they involve a pop of the top stack element, and pushing any number, including zero, of new elements. The challenge is in constructing the correct $k+1$-parameter instantiation of `C`, from the current parameters of the type. Each of these $k+1$ is also an instantiation of `C` which may require more such parameters. Even though the number of ingredients is small, the resulting type expressions tend to be excessively long and unreadable.

The predicament is ameliorated a bit by the idea, demonstrated above with auxiliary type `P'` (Figure 12), of delegating the task of creating a complex type to an auxiliary generic type. The task of this sidekick is simplified if some of its generic parameters are sub-expressions that recur in the desired result.

Cases in point are $\delta(\gamma_1,\sigma_1) = \mathsf{push}(\gamma_1,\gamma_1,\gamma_2)$, and $\delta(\gamma_2,\sigma_1) = \mathsf{push}(\gamma_2,\gamma_2)$ of Table 1. The corresponding sidekick types, ($\gamma$`1`$\sigma$`1_Push_`$\gamma$`1`$\gamma$`1`$\gamma$`2` and $\gamma$`2`$\sigma$`1_Push_`$\gamma$`2`$\gamma$`2`) can be found in lines 36–43 of Figure 15. The first of these define the correct return type of $\sigma$`1()` in case $\gamma$`1` is the top element, the second of $\sigma$`2`, in case $\gamma$`2` is the top element. Examine now the definition of types `C`$\gamma$`1`,`C`$\gamma$`2` in the figure, and in particular lines 21–23 and 29–31 which define the list of types they extend. Notice that each extends one of the sidekicks, inheriting the covariant overrides of $\sigma$`1()`.

More generally, economy of expression may require that for each case of $\delta(\gamma,\sigma) = \mathsf{push}(\zeta)$ in the transition table, one creates a sidekick type which overrides the appropriate $\sigma$`()` function. The appropriate `C`$\gamma$ type then inherits the definition from the sidekick.

### Conclusion

The proof of Theorem 1 is an algorithm, taking as input some jDPDA, and returning as output a set of Java type definitions. The returned types, allow a call chain $\mathsf{java}(\alpha)$, such that the type of the returned object represents the configuration of the input automaton after reading $\alpha$. If the automaton rejects after $\alpha$, then the returned type is the illegal $\Sigma\Sigma$, and if the automaton accpets, the type shall be $L$.

## 8 The Prefix Theorem

▶ **Theorem 2.** *Let $A$ be a DPDA recognizing a language $L \subseteq \Sigma^*$. Then, there exists a* Java *type definition, $J_A$ for types* `L`, `A`, `C` *and other types such that the* Java *command*

```
C c = A.build.java(α);
```
(3)

*type checks against $J_A$ if an only if there exists $\beta \in \Sigma^*$ such that $\alpha\beta \in L$ and type* `C` *is the configuration of $A$ after reading $\alpha$. Furthermore, for any such $\beta$, Theorem 1 applies such that the* Java *command*

```
L ℓ = A.build.java(αβ).$();
```
(4)

*always type-checks. Finally, the program $J_A$ can be effectively generated from $A$.*

Informally, a call chain type-checks if and only if it is a prefix of some legal sequence. Alternatively, a call chain won't type-check if there is no continuation that leads to a legal string in $L$.

The proof resembles that of Theorem 1. We provide a similar implementation for a jump-stack [19], that will not compile under illegal prefixes.

[19] recall that the two formal constructs have the same expressive power

```
class A { // Encode automaton A
  private static class ΣΣ    // Encodes set Σ*, type of reject
    { /* empty */  }
  static class L extends ΣΣ // Encodes set L ⊆ Σ*, type of accept
    { /* empty */  }
  // Configuration of the automaton
  interface C<       // Generic parameters:
    Rest extends C, // The rest of the stack, for pop or jump(γ) operations
    JRγ1 extends C, // Type of  Rest.jump(γ1), may be rest, or anything in it.
    JRγ2 extends C  // Type of  Rest.jump(γ2), may be rest, or anything in it.
  >
  {
    ΣΣ $();          // δ transition on end of input; invalid language by default
    C σ1();          // δ transition on σ1; dead end by default
    C σ2();          // δ transition on σ2; dead end by default
    C σ3();          // δ transition on σ3; dead end by default
    public interface E extends C<¤,¤,¤> { /* Empty stack configuration */ }
    interface ¤ extends C<¤,¤,¤> { /* Error configuration. */ }
    interface Cγ1< // Configuration when γ1 is at top
      Rest extends C, JRγ1 extends C, JRγ2 extends C
    > extends
      C<Rest, JRγ1, JRγ2>
      ,γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,Cγ1<Rest, JRγ1, JRγ2>>
    {
      @Override L $();
    }
    interface Cγ2< // Configuration when γ2 is at top
      Rest extends C, JRγ1 extends C, JRγ2 extends C
    > extends
      C<Rest, JRγ1, JRγ2>
      ,γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>
    {
      @Override Rest σ2();
      @Override JRγ1 σ3();
    }
    interface γ1σ1_Push_γ1γ1γ2<Rest extends C,JRγ1 extends C,JRγ2 extends C,
                               P extends Cγ1<Rest, JRγ1, JRγ2 >>{
      // Sidekick of δ(γ1, σ1) = push(γ1, γ1, γ2)
      Cγ2<Cγ1<P, Rest, JRγ2>,P,JRγ2> σ1();
    }
    interface γ2σ1_Push_γ2γ2<Rest extends C,JRγ1 extends C,JRγ2 extends C>{
      // Sidekick of δ(γ2, σ1) = push(γ2, γ2)
      Cγ2<Cγ2<Rest, JRγ1, JRγ2>, JRγ1, Rest> σ1();
    }
  }
  static Cγ1<E,¤,¤> build = null;
}
```

**Figure 15** Type encoding of jDPDA $A$ (as defined in Table 1).

The main difference between the two theorems is: in Theorem 1 we allowed illegal call chains to compile, but not return the required `L` type, while in Theorem 2 the illegal chain won't compile at all.

Since the code suggested by the proof is similar to the code presented above, only the differences will be discussed.

We will use the same running example, defined by Table 1.

## 8.1 Main Types

The main types here are a subset of the previously defined main types.

```
static class L // Encodes set L ⊆ Σ*, type of accept
  { /* empty */  }
public interface E { /* Empty stack configuration */ }
interface ¤ { /* Error configuration. */ }
```

First, type ΣΣ is removed. A call chain that doesn't represent a valid prefix won't compile, thus, there is no need for an error return type such as ΣΣ. Second, `interface C` is

```
static void accepts() {
  A.build.$();
  A.build.σ1().σ3().$();
  A.build.σ1().σ2().$();
  A.build.σ1().σ1().σ2().σ3().σ1().σ2().$();
}
static void rejects() {
  A.build.σ1().$();
  A.build.σ2();
  A.build.σ1().σ2().σ3();
  A.build.σ1().σ1().σ2().σ3().σ1().$();
}
```

**Figure 16** Accepting and non-accepting call chains with the type encoding of jDPDA $A$ (as defined in Table 1). All lines in `accepts` type-check, and all lines in `rejects` cause type errors.

removed. Without it, the configuration types won't have the methods $\sigma$`1()`, ... ,$\sigma k$`()` and `$()` from the supertype. These inherited methods, is what differentiates the previous proof from the current. Classes `¤` and `E` are defined similarly, except now they don't extend any type.

## 8.2 Top-of-Stack Types

Types `C`$\gamma$`1`, ... ,`C`$\gamma k$, still represent stacks with $\gamma$`1`, ... ,$\gamma k$ as their top element, this time, the methods are defined ad-hock, in each type (they are not added in this figure as they are added with the use of sidekicks). In $A$ there are two such types:

```
interface Cγ1< // Configuration when γ1 is at top
  Rest, JRγ1, JRγ2
> extends
  γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,Cγ1<Rest, JRγ1, JRγ2>>
{
}
interface Cγ2< // Configuration when γ2 is at top
  Rest, JRγ1, JRγ2
> extends
  γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>
{
}
static Cγ1<E,¤,¤> build = null;
```

Note, that the type parameters of the former types hasn't changed, since the model we are trying to implement, hasn't changed. These $k+1$ parameters still suffice for our cause.

In Figure 16, call chains in the `accepts()` method correctly type-checks (i.e., in $L$), while the chains in `rejects()` do not type-check (i.e., these prefixes have no continuation that can lead to a legal word in $L$), where the last method invocation generates an

"method ... is undefined for the type ... "

error message.

The main difference between Figure 16 and Figure 14 is that there is no need to use an auxiliary function `isL()` as in Figure 16 since now illegal prefixes do not type-check.

## 8.3 Transitions

Due to the changes we expressed, the transition table is encoded slightly different.

Encoding of the legal operations accept, jump($\gamma_i$) and push($\zeta$) remains as in Theorem 1, since we want the same behavior for legal call chains. The minor differences are in the illegal operations reject and $\perp$:

```
1 static class A { // Encode automaton A
2   static class L // Encodes set L ⊆ Σ*, type of accept
3     { /* empty */  }
4   public interface E { /* Empty stack configuration */ }
5   interface ¤ { /* Error configuration. */ }
6   // Configuration of the automaton
7   interface Cγ1< // Configuration when γ1 is at top
8     Rest, JRγ1, JRγ2
9   > extends
10    γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,Cγ1<Rest, JRγ1, JRγ2>>
11  {
12    L $();
13  }
14  interface Cγ2< // Configuration when γ2 is at top
15    Rest, JRγ1, JRγ2
16  > extends
17    γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>
18  {
19    Rest σ2();
20    JRγ1 σ3();
21  }
22  interface γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,P extends Cγ1<Rest, JRγ1, JRγ2 >>{
23    // Sidekick of δ(γ1, σ1) = push(γ1, γ1, γ2)
24    Cγ2<Cγ1<P, Rest, JRγ2>,P,JRγ2> σ1();
25  }
26  interface γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>{
27    // Sidekick of δ(γ2, σ1) = push(γ2, γ2)
28    Cγ2<Cγ2<Rest, JRγ1, JRγ2>, JRγ1, Rest> σ1();
29  }
30  static Cγ1<E,¤,¤> build = null;
31 }
```

**Figure 17** Type encoding of jDPDA $A$ (as defined in Table 1) that allow a partial call chain, if and only if, there exists a legal continuation, that leads to a word in $L$ (the language of $A$).

**reject** Since we add the methods ad-hock to each type, the reject entry means that the corresponding type, *won't* have a `$()` method, i.e., type `Cγ2` doesn't have a method `$()`.

$\perp$ We encounter $\perp$ on the transition function when some input character $\sigma$ is not allowed for the top of the stack element $\gamma$. In that case, the corresponding type `Cγ` *must not* have a method for $\sigma$, this way, invoking the methods will result in type error. In Table 1 a $\perp$ may occur when the top of the stack is $\gamma_1$ and the input character is $\sigma_2$, thus, no method `σ2` is introduced in type `Cγ1`.

The use of sidekicks is still allowed and recommended to improve readability of code.

### Conclusion

In this section, a proof, similar to the one in Section 7 is provided. An algorithm was introduced, to not only emulate the running of some jDPDA $A$, but also to "halt it" in the earliest time possible, i.e., only if there is no legal call chain from this point to result in a legal word in the language of $A$.

## 9 Notes on Practical Applicability

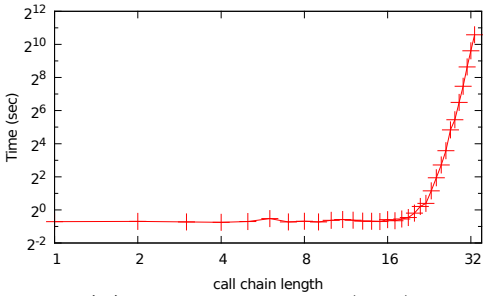
Theorem 1 and its proof above provide a concrete algorithm for converting an EBNF specification of a fluent API into its realization:

1. Convert the specification into a plain BNF form [20].

[20] `http://lampwww.epfl.ch/teaching/archive/compilation-ssc/2000/part4/parsing/node3.html`

```
static interface Cons<Car,Cdr>{
  Cons<
    Cons<Car,Cdr>,
    Cons<Car,Cdr>
  > d();
}
```

**(a)** Encoding of a binary type tree




**(b)** Compilation time (sec[a]) *vs.* length of call chain.

[a] measured on an Intel i5-2520M CPU @ 2.50GHz ×4, 3.7GB memory, Ubuntu 15.04 64-bit, `javac` 1.8.0_66

**Figure 18** Exponential compilation time for a simple JAVA program.

2. Convert this BNF into a type of DPDA (using parsing algorithms e.g., LR(k), LALR, LL(k)). This conversion might fail [21].
3. Convert this DPDA into a jDPDA. (Conversion is guaranteed to succeed)
4. Apply the proof to generate appropriate JAVA type definitions, making sure to augment methods with code to maintain the fluent-call-list. Parsing the fluent-call-list can be done either in each method, or lazily, when the product of the fluent API call chain is to be used.

Although possible, a practical tool that uses the proof directly is a challenge. Part of the problem is the complexity of the algorithms used, some of which, e.g., the DPDA and jDPDA equivalence have never been implemented. Yet another issue that clients of compiler-compiler have grown to expect facilities such as means for resolving ambiguities, manipulation of attributes, etc. Also, for a fluent API to be elegant and useful, it should support method with parameters whose parameters are also defined by a fluent API: these two APIs may mutually recursive and even the same. Support of these features through four or so algorithmic abstractions may turn out to be a decent engineering task.

Yet another challenge is controlling the compiler's runtime. Learning that linear time parsers and lexical analyzers are possible, and being accustomed to seeing these in practice, one may expect the compiler would run in linear, or at least polynomial time. As it turns out, this time is exponential in the worst case (at least for `javac`). An encoding of a S-expression in type `Cons` (Figure 18(a)) is a not terribly complex such worst case.

Type `Cons` takes two type parameters, `Car` and `Cdr` (denoting left and right branches). Denote the return type of `d()` by

$$\tau = \texttt{Cons< Cons<Car, Cdr>, Cons<Car, Cdr> >}.$$

Let $\sigma$ denote the type of the `this` implicit parameter to `d`. Now, since $\tau = \texttt{Cons<}\sigma,\sigma\texttt{>}$, we have $|\tau| \geq 2|\sigma|$, where the size of a type is measured, e.g., in number of characters in its textual representation. Therefore, in a chain of $n$ calls to `d()`

$$\texttt{(Cons<?,?>(null)).}\overbrace{\texttt{d().}\cdots\texttt{.d()}}^{n\text{ times}}\texttt{;} \tag{5}$$

the size of the resulting type is $O(2^n)$.

[21] In the LR case, we know [30] there exists an equivalent grammar for which the conversion will succeed

Figure 18(b) shows, on the doubly logarithmic plane, the runtime (on a Lenovo X220) of the `javac` compiler (version 1.8.0_66) in face of a Java program assembled from Figure 18 and Equation 5 placed as the single command of `main()`. Exponential growth is demonstrated by the right-hand side of the plot, in which curve converges on a straight line. (In fact, a variation of the construction may lead to even super-exponential growth rate of the size of types.)

We believe that this exponential growth is due to a design flaw in the compiler. Had the compiler used a representation of types that allows sharing of expression types, compilation time would be linear.

Still, with current compiler technology, the type encoding scheme demonstrated in Figure 15 might not be scalable.

## 10 Conclusion and Future Work

The main contribution of this work is the proof that most useful grammars have a fluent API. This brings good news to library designers laboring at making their API slick, accessible, and more robust to mistakes of clients: If your API can be phrased in terms of a "decent" BNF, do not lose hope; the task may be Herculean, but it is (most likely) possible.

Other practitioners may appreciate the toolbox of type encodings offered here gaining better understanding of the computational expressiveness of Java generics and type hierarchy, and, a better tool for designing, experimenting with and perfecting fluent APIs.

However, once possibility was demonstrated theoretically, the next *research* challenge is in an actual fluent API compiler based on lighter weight parsing algorithms. Precisely, the challenge is in developing a parsing (or at least recognition) algorithm which is not only efficient but also falls within the limited computing power of the Java types.

On the theoretical front, one may ask whether our result is the best possible: Can the Java type system be coerced to recognize general (that is, nondeterministic) context-free languages?

As mentioned earlier, to the best of our knowledge, the complexity of the Java type checker has never been analyzed. In light of the empirical finding in Section 9, research in this direction may be worthwhile.

Other directions include formalizing the proof in Section 7 here and extensions to other languages.

On a philosophical perspective, several modern programming languages acquire high-level constructs at a staggering rate (C++ and Scala [35] being prominent examples). The main yardstick for evaluation these is "programmer's convenience". This work suggests an orthogonal perspective, namely computational expressiveness, or, stated differently, ranking of a new construct by its ability to recognize languages in the Chomsky hierarchy [12].

## References

1 David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ in Depth Series. Addison-Wesley, 2004.

2 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

3 Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In Gary Leavens, editor, *Proc. of the 24th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'09)*, pages 1015–1022, Orlando, FL, USA, October 2009. ACM

Press. URL: http://doi.acm.org/10.1145/1639950.1640073, doi:10.1145/1639950.1640073.

4 Ken Arnold and James Gosling. *The* Java *Programming Language*. The Java Series. Addison-Wesley, Reading, MA, 1996.

5 Matthew H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley, 1998.

6 Jean-Michel Autebert, Jean Berstel, and Luc Boasson. *Context-Free Languages and Pushdown Automata*. Springer, 1997.

7 Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming. In *Advanced Functional Programming*, pages 28–115. Springer, 1999.

8 Nels E. Beckman, Duri Kim, and Jonathan Erik Aldrich. An empirical study of object protocols in the wild. In Mira Mezini, editor, *Proc. of the 25th Euro. Conf. on OO Prog. (ECOOP'11)*, volume 6813 of *LNCS*, pages 2–26, Lancaster, UK, June25-29 2011. Springer.

9 Kevin Bierhoff and Jonathan Erik Aldrich. Lightweight object specification with typestates. In Michel Wermelinger and Harald C. Gall, editors, *Proc. of the 10th European Soft. Eng. Conf. and 13th ACM SIGSOFT Symp. on the Foundations of Soft. Eng. (ESEC/FSE'05)*, pages 217–226, Lisbon, Portugal, September 2005. ACM Press.

10 Eric Bodden. TS4J : A fluent interface for defining and computing typestate analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in* Java *Program Analysis - SOAP '14*, pages 1–6, 2014. URL: http://dl.acm.org/citation.cfm?doid=2614628.2614629http://www.bodden.de/pubs/bodden14ts4j.pdf, doi:10.1145/2614628.2614629.

11 Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Bjørn N. - Benson Freeman and Craig Chambers, editors, *Proc. of the 13th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'98)*, pages 183–200, Vancouver, BC, Canada, October18-22 1998. ACM SIGPLAN Notices 33(10).

12 Noam Chomsky. *Formal properties of grammars*. Addison-Wesley, 1963.

13 John Cocke. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 1969.

14 Bruno Courcelle. On jump-deterministic pushdown automata. *Math. Sys. Theory*, 11:87–109, 1977.

15 James C. Dehnert and Alexander Stepanov. Fundamentals of generic programming. In *Generic Programming*, pages 1–11. Springer, 2000.

16 Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000. URL: http://portal.acm.org/citation.cfm?doid=352029.352035, doi:10.1145/352029.352035.

17 Charles Donnelly and Richard Stallman. *Bison*, 2015.

18 Jay Earley. An efficient context-free parsing algorithm. *Comm. of the ACM*, 13(2):94–102, 1970. doi:10.1145/362007.362035.

19 Sebastian Erdweg, Lennart C.L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Sugarj: Library-based language extensibility. In Kathleen Fisher, editor, *Proc. of the 26th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'10)*, pages 187–188, Portland OR, USA, October22-27 2011. ACM Press.

20 Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in Java. In Peri L. Tarr and William R. Cook, editors, *Proc. of the OOPSLA'06 Companion*. ACM Press, October22-26 2006.

21 Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In Ron Crocker and Guy L. Steele Jr., editors, *Proc. of the 18th Ann. Conf. on OO Prog. Sys., Lang., &*

*Appl. (OOPSLA'03)*, pages 115–134, Anaheim, CA, USA, October 2003. ACM SIGPLAN Notices 38 (11). URL: http://www.informatik.uni-trier.de/~ley/db/conf/oopsla/oopsla2003p.html.

22 Joseph Gil and Zvi Gutterman. Compile time symbolic derivation with C++ templates. In *Proc. of the USENIX C++ Conf.*, pages 249–264, Santa Fe, NM, April 1998. USENIX Association.

23 Joseph (Yossi) Gil and Keren Lenz. Simple and safe SQL queries with templates. In Charles Consel, editor, *Proc. of the 6th Conf. on Generative Prog. & Component Eng.*, LNCS, pages 13–24, Salzburg, Austria, October 2007. ACM Press.

24 Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

25 Zvi Gutterman. Turing templates—on compile time power. Master's thesis, Technion—Israel Institute of Technology, 2003.

26 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd edition, 2001.

27 Claus Ibsen and Jonathan Anstey. *Camel in action*. Manning Publications Co., Shelter Island, NY, 2010.

28 JBoss Group. Hibernate product homepage. http://www.hibernate.org/, 2006.

29 Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for JAVA. *Proceedings of the 6th international symposium on Principles and practice of programming in* JAVA- *PPPJ '08*, 2008. doi:10.1145/1411732.1411758.

30 Donald Ervin Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. doi:10.1016/S0019-9958(65)90426-2.

31 Robert Larsen. Fluenty: A type safe query API. Master's thesis, University of Oslo, 2012.

32 Peter Linz. *An Introduction to Formal Languages and Automata*. Jones & Bartlett Learning, 2011.

33 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (ICMD'2006)*, Chicago, Illinois, 2006.

34 David R. Musser and Alexander A. Stepanov. Generic programming. In *Symbolic and Algebraic Computation*, pages 13–25. Springer, 1989.

35 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

36 Matthew M Papi. Practical pluggable types for JAVA. Master's thesis, Massachusetts Institute of Technology, 2008. URL: http://portal.acm.org/citation.cfm?doid=1390630.1390656, doi:10.1145/1390630.1390656.

37 Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

38 David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

39 Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

40 Daniel H. Younger. Recognition and parsing of context-free languages in time $n^3$. *Information and Control*, 10(2):189–208, 1967. URL: http://www.sciencedirect.com/science/article/pii/S001999586780007X, doi:10.1016/S0019-9958(67)80007-X.